Performance Analysis and Optimization of Automatic Speech Recognition

Hamid Tabani, Jose-Maria Arnau, Jordi Tubella, and Antonio González, *Fellow, IEEE*

Abstract—Fast and accurate Automatic Speech Recognition (ASR) is emerging as a key application for mobile devices. Delivering ASR on such devices is challenging due to the compute-intensive nature of the problem and the power constraints of embedded systems. In this paper, we provide a performance and energy characterization of Pocketsphinx, a popular toolset for ASR that targets mobile devices. We identify the computation of the Gaussian Mixture Model (GMM) as the main bottleneck, consuming more than 80 percent of the execution time. The CPI stack analysis shows that branches and main memory accesses are the main performance limiting factors for GMM computation. We propose several software-level optimizations driven by the power/performance analysis. Unlike previous proposals that trade accuracy for performance by reducing the number of Gaussians evaluated, we maintain accuracy and improve performance by effectively using the underlying CPU microarchitecture. First, we use a refactored implementation of the innermost loop of the GMM evaluation code to ameliorate the impact of branches. Second, we exploit the vector unit available on most modern CPUs to boost GMM computation, introducing a novel memory layout for storing the means and variances of the Gaussians in order to maximize the effectiveness of vectorization. Third, we compute the Gaussians for multiple frames in parallel, so means and variances can be fetched once in the on-chip caches and reused across multiple frames, significantly reducing memory bandwidth usage. We evaluate our optimizations using both hardware counters on real CPUs and simulations. Our experimental results show that the proposed optimizations provide 2.68x speedup over the baseline Pocketsphinx decoder on a high-end Intel Skylake CPU, while achieving 61 percent energy savings. On a modern ARM Cortex-A57 mobile processor our techniques improve performance by 1.85x, while providing 59 percent energy savings without any loss in the accuracy of the ASR system.

Index Terms—Automatic speech recognition, Gaussian mixture models, vectorization

1 Introduction

Automatic Speech Recognition (ASR) is becoming increasingly ubiquitous, significantly changing the way users interact with computers. ASR technology is at the heart of popular voice-based user interfaces such as Apple Siri [1], Microsoft Cortana [2] or Google Now [3]. These applications deliver real-time, large vocabulary, speaker independent speech recognition. However, supporting accurate real-time ASR comes at a high energy cost. To illustrate this problem, Fig. 1 shows the Real Time Factor (RTF[1]) versus Word Error Rate (WER[2]) for different configurations of Pocketsphinx [4], a widely used open source toolset for ASR, running on an ARM Cortex-A57 mobile CPU. As it can be seen, increasing accuracy causes a huge slowdown: the continuous acoustic model reduces error by 5 percentage points with respect to the simpler Phonetic Tied-Mixture (PTM) [5] model, while producing a slowdown of 2.4x and increasing energy consumption by 2.78x. Note that this is not only the case for Pocketsphinx, as other ASR toolsets exhibit similar power/performance trade-offs [6].

Previous solutions improve ASR performance by reducing the amount of computation required to convert the speech signal to words. One commonly used strategy is to simplify the acoustic model. Phonemes are typically modeled by using Gaussian Mixture Models (GMMs). The semi-continuous and PTM acoustic models of Pocketsphinx, included in Fig. 1, significantly constrain the number of Gaussians in each mixture to improve performance. Although highly effective, this approach reduces accuracy to a large extent, as simple acoustic models cannot capture the complexity of speech.

Recognizing that accuracy is probably the most important parameter in an ASR system, we take a completely different approach as we improve performance while achieving the same accuracy of the baseline configuration. We boost GMM computation performance by applying low-level optimizations to the software in order to maximize the usage of the available CPU resources.

The software optimizations presented in this paper are based on an extensive analysis of the power/performance behavior of Pocketsphinx. Fig. 2 summarizes the results of the analysis. As it can be seen, the Gaussian Mixture Model evaluation, i.e., the acoustic model, is the main bottleneck consuming more than 80 percent of the execution time.

1. RTF is the ratio between execution time and the duration of the utterance (lower RTF is faster).

2. WER is defined as the number of insertions plus deletions plus substitutions that are required to convert the recognized word sequence into the reference word sequence, divided by the total number of words of the utterance.

- *The authors are with the Computer Architecture Department, Universitat Politècnica de Catalunya, Barcelona 08034, Spain. E-mail: {htabani, jarnau, jordit, antonio}@ac.upc.edu.*

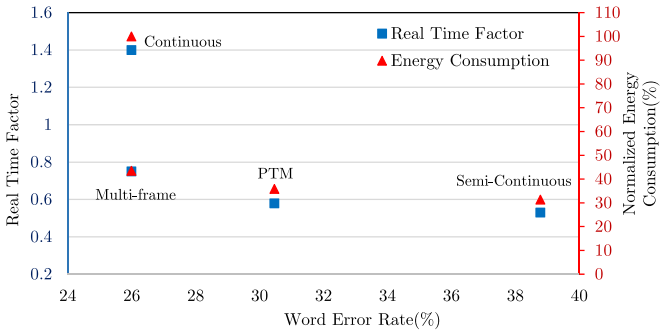
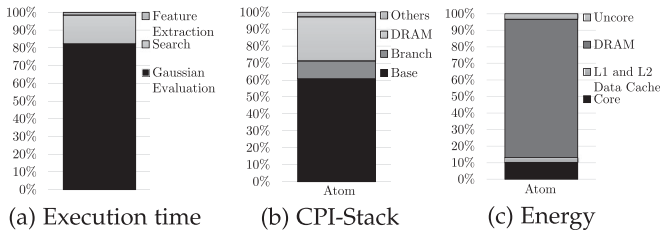


Fig. 1. Real Time Factor and Normalized Energy consumption versus Word Error Rate for different acoustic models in Pocketsphinx running on an ARM Cortex-A57 mobile CPU. Multi-frame shows our optimized decoder. Energy consumption for different acoustic models are normalized with respect to the continuous acoustic model.



(a) Execution time (b) CPI-Stack (c) Energy

Fig. 2. Summary of results for power/performance analysis of Pocketsphinx.

Furthermore, the CPI stack for an Atom-like processor shows that the main sources of stalls in the CPU are branch mispredictions and accesses to main memory. Finally, the power breakdown for the same CPU clearly shows that the DRAM is the main energy consumer.

Our software optimizations target the problems identified during the analysis. Regarding branch misprediction penalties, we show how the GMM evaluation code can be refactored to remove the most critical branches. As regards to the DRAM, we propose a multi-framing scheme where means and variances of a Gaussian are fetched once in the CPU caches and reused for evaluating the Gaussian in multiple frames of speech, improving the locality of memory accesses.

The main hurdle for implementing multi-framing in modern ASR systems is the interaction with lazy GMM evaluation. Due to the huge search space, ASR systems employ aggressive pruning to achieve real-time performance by dynamically discarding unlikely interpretations of the speech signal. Because of the pruning, only a subset of the Gaussians is active for a given frame of speech whereas the likelihoods of the other, inactive, Gaussians are not required. Pocketsphinx employs lazy GMM evaluation to avoid computing and fetching from memory inactive Gaussians. Combining lazy GMM evaluation with our multi-framing scheme is challenging as only the active Gaussians for the first frame in the batch are known. In this paper, we propose a novel prediction scheme of active Gaussians that is highly effective and allows the use of both lazy GMM evaluation and multi-framing, substantially reducing main memory bandwidth usage.

Finally, we introduce SIMD instructions in Pocketsphinx to improve the performance and energy efficiency of the GMM computation. Furthermore, we propose a novel memory layout to store the means and variances that increases the amount of vectorizable code. Fig. 1 shows that our optimized decoder, labeled as *Multi-frame*, provides the same accuracy than the continuous acoustic model while achieving performance and energy consumption close to the simpler PTM and semi-continuous acoustic models.

This paper focuses on low-power, real-time ASR. Its main contributions are the following:

- We provide a detailed analysis of the performance/power behavior of a CPU when running Pocketsphinx, identifying branch mispredictions and main memory accesses as the main bottlenecks and the DRAM as the main source of energy consumption.
- We used a refactored GMM evaluation code to remove the most critical branches, improving performance by 13 percent and reducing energy by 12.5 percent.
- We vectorize the main loop of the GMM evaluation to use SIMD instructions. We also introduce a novel memory layout to store the means and variances of the Gaussians that increases the amount of vectorizable code. The use of the Vector Processing Unit (VPU) improves performance by 73 percent and reduces energy by 41 percent.
- We propose a multi-framing GMM evaluation technique that computes the Gaussians for multiple frames in advance. Furthermore, we present a novel prediction scheme of active Gaussians to combine multi-framing with lazy GMM evaluation. This technique reduces memory bandwidth by 57 percent, providing 34 percent speedup and 25.5 percent energy savings.
- Overall, all the above optimizations provide 2.68x and 61 percent energy savings on an Intel Skylake CPU. On a low-power ARM A57 CPU, our optimized decoder achieves 1.85x speedup and 59 percent reduction in energy consumption.

The remainder of this paper is organized as follows. The next section provides background information on ASR systems. Section 3 presents the results of the power/performance analysis of Pocketsphinx. Section 4 presents our software optimizations for GMM computation and Section 5 describes our methodology. Section 6 discusses the performance and energy results. Section 7 reviews related work and, finally, Section 8 sums up our conclusions.

2 Background

The objective of speech recognition is the transcription of acoustic signals into a sequence of words. A state-of-the-art ASR pipeline consists of three stages: *Feature Extraction*, *Acoustic Model* and *Search Engine* as it is shown in Fig. 3. This pipeline works as follows. First, the input audio signal is split in frames, where each frame represents a 10 ms interval of the input signal. Next, the *Feature Extraction* component transforms the audio samples within a frame into a vector of features. The features are then converted into a sequence of phonemes by the *Acoustic Model*. Pocketsphinx *Acoustic Model* is based on Gaussian Mixture Models [7], i.e., each phoneme is modeled as a mixture of Gaussian functions. Finally, the *Search Engine* transforms the sequence

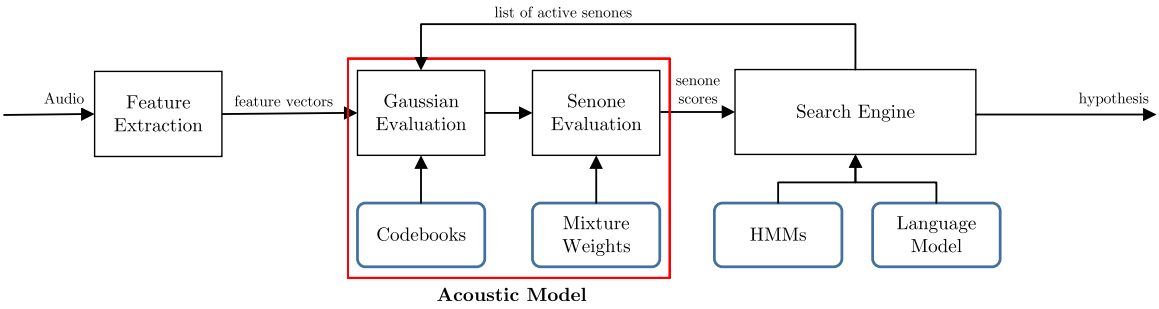


Fig. 3. Pocketsphinx decoder architecture.

of phonemes into a sequence of words by executing the Viterbi beam search [7], [8], [9] on a pre-compiled Hidden Markov Model (HMM) [7].

The *Acoustic Model* is known to be the most expensive part of an ASR system [10], [11]. Our power/performance analysis, summarized in Fig. 2, also supports this claim.

2.1 Acoustic Model

ASR systems create acoustic models for sub-word units or phonemes. Due to co-articulation effects, the production of sound corresponding to a phoneme is influenced by neighbouring phonemes. Context-dependent phones or triphones [12] are typically employed by ASR systems, including Pocketsphinx, as they are able to model such variations. There are only around 50 phonemes in spoken English, so there are around 50^3 triphones, but only a fraction of them are observed in practice.

Triphones are modeled in Pocketsphinx by using Gaussian Mixture Models. The gth component of the mth GMM is a normal distribution with mean vector $\mu_{m,g}$ and standard deviation vector $\sigma_{m,g}$. Each mixture component also has a scalar mixture coefficient or weight $w_{m,g}$. Hence, the probability of observing a given frame of speech with feature vector x in the GMM m is given by Equation (1), where g ranges over the number of Gaussians in the mixture. The expression $\mathcal{N}(\cdot)$ is the value of the Gaussian density function at x. For numerical stability, the multivariate Gaussian distribution $\mathcal{N}(\cdot)$ is computed in log-space by using Equation (2). The dimensionality of the Gaussians is equal to the dimensionality of the feature vector x. During the recognition process Equation (1) has to be evaluated for every GMM on a frame basis. In these equations, D, M and N are representing determinant, dimensionality of the feature vector and number of Gaussians respectively

$$GMM_m(x) = \sum_{g=0}^{N-1} w_{m,g}\mathcal{N}(x, \mu_{m,g}, \sigma_{m,g}) \quad (1)$$

$$\mathcal{N}(x, \mu_{m,g}, \sigma_{m,g}) = D_{m,g} - \sum_{c=0}^{M-1} \frac{(x_c - \mu_{m,g,c})^2}{2\sigma^2_{m,g,c}}. \quad (2)$$

In a fully continuous acoustic model each triphone has its own separate weighted GMM. However, computing such a big number of Gaussian functions is completely unfeasible for real-time speech recognition. In practice, multiple triphones share the same GMM to reduce the computational cost of the *Acoustic Model*. In the continuous model of Pocketsphinx triphones are grouped into clusters called senones. All the triphones that belong to the same senone share the same GMM. The latest generic acoustic model of Pocketsphinx (en-us-5.2) has 5,138 senones. Although the continuous model offers the highest accuracy in Pocketsphinx, other acoustic models are included in an attempt to reduce the amount of computation. The PTM model has a GMM for each context-independent phoneme or basephone. Triphones that belong to the same basephone share the same GMM. Hence, the number of GMMs is reduced from 5,138 in the continuous model to 42 in PTM, which is the number of basephones in this model. On the other hand, the semi-continuous acoustic model employs just one GMM that is shared by all the triphones, but each triphone has its own mixture coefficients, i.e., the weights applied to each Gaussian ($w_{m,g}$ in Equation (1)) are different.

On the other hand, not all the senones are active for a given frame of speech, as some of them may be pruned away during the search process. Pocketsphinx only computes the GMM for active senones, as the acoustic likelihood for the other, inactive, senones is not required for the search. By doing this, the workload for the continuous acoustic model is reduced by approximately one half. In order to implement this optimization the *Search Engine* generates a list of active senones based on the result of the pruning. The *Acoustic Model* uses this feedback to avoid GMM computation for senones that are inactive.

In this paper we use the latest continuous acoustic model of Pocketsphinx (en-us-5.2) to analyze CPU power/performance and evaluate our optimizations. Note that the proposed techniques are generally applicable to any acoustic model based on GMMs, independently of the specific parameters such as the number of senones or the number of Gaussians.

3 Energy-Performance Analysis

This section provides a detailed power/performance analysis of a CPU when running Pocketsphinx with the continuous acoustic model presented in Section 2.1. First, we describe the bottlenecks in the software and the main sources of stalls in the CPU pipeline. Second, we relate those CPU pipeline stalls with the source code of the GMM computation, identifying the branch instructions and memory accesses that cause such stalls. Finally, we complete the analysis with an energy characterization of Pocketsphinx.

3.1 Performance Characterization

We first profile the execution time of the different stages in an ASR pipeline. We run Pocketsphinx on an ARM mobile

TABLE 1
Hardware Parameters Employed for the Simulations

Core	Atom-like OoO, 2-wide, 1.33 GHz
L1-D Cache	24 KB, 6-way, 64 B lines, LRU
L2 Cache	1 MB, 16-way, 64 B lines, LRU
Main Memory	4 GB, 12.8 GB/s bandwidth
Branch predictor	Pentium M branch Predictor 15 cycles misprediction penalty
Technology	22 nm
Prefetchers	G HB, prefetch-degree = 2, table-size = 512

CPU with the parameters shown in Table 3. The results are provided in Fig. 2a. The *Acoustic Model* takes up the bulk of execution time, as it requires 82.4 percent of the total time to convert the speech into words. On the other hand, the *Search Engine* and the *Feature Extraction* take 16.1 and 1.42 percent of execution time respectively. Therefore *Acoustic Model* is clearly the main bottleneck.

Fig. 2b shows the CPI stack for an Atom-like processor, whose parameters are provided in Table 1, when executing the *Acoustic Model*. We run Pocketsphinx on the Sniper simulator to generate the CPI stack. Accesses to off-chip system memory (DRAM) and branch mispredictions (Branch) are the main sources of stalls in the CPU pipeline. A more detailed analysis of the GMM evaluation code provides more insights on the sources of such CPU stalls.

Listing 1. C-like pseudocode for acoustic model computation.

```
1.  void GMM(int m, float *x, float *out) {
2.    for (i = 0; i < TOP_N_GAU; i++)
3.      out[i] = WORST_VAL;
4.    for (g = 0; g < NUM_GAUSSIANS; g++) {
5.      float val = det[m][g];
6.      for (c = 0; c < NUM_COMPONENTS; c++) {
7.        float diff = x[i] - means[m][g][c];
8.        val -= diff * diff * vars[m][g][c];
9.        if (val < gauval[TOP_N_GAU - 1])
10.         break; //Not in TOP N
11.     }
12.     if (val >= gauval[TOP_N_GAU - 1])
13.       InsertInSortedList(out, val);
14.   }
15. }
```

Listing 1 shows the *Acoustic Model* implementation in Pocketsphinx. The function *GMM* is called for every senone, i.e., Gaussian mixture, on a frame basis. *GMM* evaluates the different Gaussian functions in the mth mixture for a given feature vector x, implementing the computation described in Equation (2) with a few optimizations that work as follows. First, all the computations that do not depend on the input feature vector x are pre-computed offline. The determinant vector, *det*, stores the value of $D_{m,g}$ for every Gaussian in the mixture. In a similar way, the matrix *vars* (short for variances) stores the result of $1/2\sigma_{m,g,c}$ for every component in the mixture. Regarding the second optimization, in an attempt to reduce the amount of computation only the N Gaussians with highest likelihood are used to compute the senone scores. The default value of N is 4 in Pocketsphinx. Therefore, the top 4 Gaussians are selected in *GMM* and only those four will be used to compute the final score. However, in order to select the top 4 Gaussians all the Gaussians in the mixture have to be computed. Nevertheless, computing all the components for each Gaussian might not be necessary as the likelihood is a continuously decreasing function. So if the likelihood of a Gaussian becomes smaller than the worst likelihood in the current top 4, we are sure this is not one of the 4 best Gaussians and, hence, we can stop computation for that Gaussian. Lines 9-10 of Listing 1 implement this optimization. Furthermore, lines 12-13 take care of inserting the Gaussian in the corresponding position in the sorted array of best Gaussians.

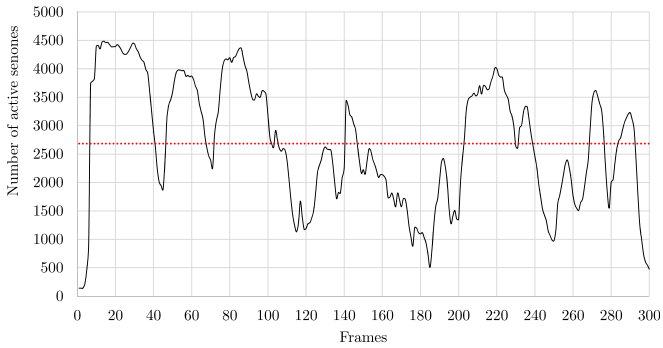


Fig. 4. Number of active senones versus frames of speech. Red dotted line shows average number of active senones per frame, which is 2,675 out of 5,138.

We computed the average number of iterations of the innermost loop in *GMM* function (lines 6-10). We found that on average 28.6 components are computed out of 36. This means a reduction of approximately 20 percent of the computation. However, this comes at the cost of having an if-sentence inside the innermost loop of the most critical function, which requires two extra x86 instructions per loop iteration to do a comparison and a branch. Moreover, we have another if-sentence after the innermost loop and extra code to insert the Gaussian in a sorted array. We found that the number of conditional branches when selecting the top 4 Gaussians increases by a factor of 1.42x with respect to a version that uses all the Gaussians. In addition, the total number of mispredicted branches increases by a factor of 1.79x.

3.2 Memory Characterization

The main source of CPU stalls is the latency of accesses to system memory, labeled as DRAM in Fig. 2b. Those memory accesses are mainly for fetching Gaussian parameters, i.e., *means* and *vars*. These parameters cannot be stored in the on-chip caches due to the big memory footprint. There are 5,138 senones in the continuous acoustic model of Pocketsphinx. Each senone has its own GMM that consists of 32 Gaussians of 36 components each one. The array *det* stores the determinant for each Gaussian, so its dimensions are $5138\ senones \times 32\ gaussians$ and it requires approximately 0.63 MBytes. The dimensionality of *means* and *vars* matrices is $5138\ senones \times 32\ gaussians \times 36\ components$, so a total of 22.6 MBytes of system memory per matrix is required. Therefore, the total memory footprint for the Gaussian parameters is 45.7 MBytes.

Only Gaussian parameters for active senones in a given frame are fetched from memory. Fig. 4 shows the number of active senones versus frames of speech. On average, 2,675

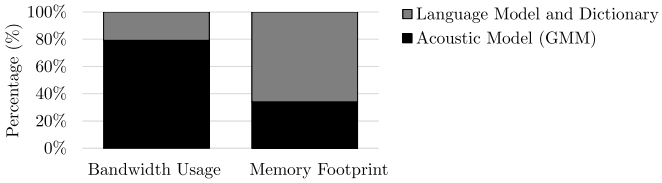


Fig. 5. Memory bandwidth usage and memory footprint in Pocketsphinx.

of the senones are active for a given frame out of 5,138, which means that 52 percent of the GMMs are evaluated per frame. This still requires a total of 23.8 MBytes of memory per frame. Even if a senone is active for a sequence of consecutive frames, the CPU cannot exploit this frame-to-frame reuse due to the big memory footprint for processing one frame of speech. Fig. 5 shows the bandwidth usage and memory footprint breakdown in Pocketsphinx. As it shows, acoustic model parameters consume less than 40 percent of the total required memory. However, as the figure shows, 79 percent of the memory bandwidth is used to fetch the GMM parameters due to the per-frame GMM evaluation. Note that the aforementioned bandwidth usage is only to fetch parameters of active senones.

The innermost loop of *GMM* function (lines 6-10 in Listing 1) includes three memory accesses for fetching the corresponding input vector x, the mean and the variance. The input vector of each frame requires just 144 bytes (36 fp elements) and exhibits high temporal reuse as it is the same for all the Gaussians. Memory accesses to *means* and *vars* matrices exploit spatial locality at the line level, so only the first access to a memory line misses in the L1. For a line size of 64 bytes, 16 elements are stored per line, so the miss ratio is 1/16. With those access patterns the miss ratio in the L1 is close to 4 percent as illustrated in Fig. 6. Spatial locality is exploited only in the L1, and there is no temporal locality for means and variances, so the miss ratios for L2 and L3 caches are close to 100 percent if prefetchers are not used. Hardware prefetchers can significantly reduce the miss ratios for L2 and L3, as illustrated in Fig. 6. This reduction in miss ratio provides 2.27x speedup. Due to their huge impact on performance, the baseline CPU configuration that we use for the experiments always employs hardware prefetchers.

Despite the good hit rate in the L1, an Atom-like processor is not able to completely hide the memory latency. Note that the FP to memory access ratio is significantly low for GMM computation. Processing a component of a Gaussian requires fetching 12 bytes from memory (vector component, mean and variance) and only 4 fp operations are performed on these data. Therefore, the FP to mem ratio is 4 flops/12 bytes = 0.33 flops per byte.

3.3 Energy Characterization

Fig. 2c shows the energy breakdown for an Atom-like CPU when running Pocketsphinx. As it can be seen, the DRAM is the main source of energy drain, consuming 83.5 percent of the total energy. The CPU represents only 10 percent of the energy. The main consumers in the CPU are the FP units and the L1 data cache, requiring 3.1 and 3 percent of the energy respectively.

The poor temporal locality, which forces memory accesses to fetch Gaussian parameters from system memory on a frame basis, is the reason why DRAM consumes most of the energy. Section 4.4 proposes a technique to improve temporal locality in order to reduce the number of off-chip memory accesses and thus save DRAM energy.

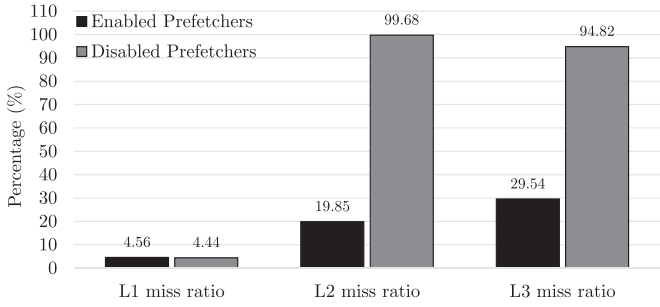


Fig. 6. Miss ratios for different levels of memory hierarchy with and without prefetchers.

4 Pocketsphinx Optimizations

This section describes our optimizations to boost GMM evaluation performance and reduce energy consumption.

Listing 2. x86 code GMM evaluation.

```
1. Top N Gaussians                  All 32 Gaussians
2. add 0x4, rax                     movss (rdi,rax,4),xmm0
3. ucomiss xmm2,xmm1                subss (rcx,rax,4),xmm0
4. jb 42a0cd                        mulss xmm0,xmm0
5. movss (rdx,rax,1),xmm0           mulss (r15,rax,4),xmm0
6. cmp rsi,rax                      add 0x1,rax
7. subss (rcx,rax,1),xmm0           cmp eax,esi
8. mulss xmm0,xmm0                  subss xmm0,xmm1
9. mulss (r15,rax,1),xmm0           jg 429fe0
10. subss xmm0,xmm1
11. jne 42a010
```

4.1 Removing Branches in GMM Evaluation

Pocketsphinx continuous acoustic model selects the top 4 Gaussians to reduce the number of iterations in the innermost loop of *GMM* function (see lines 6-10 in Listing 1) by 20.5 percent. However, it requires an additional branch instruction, with its corresponding compare instruction in x86, to branch outside the loop when a Gaussian is discarded. In other words, this optimization reduces the number of components computed, with a subsequent reduction in the number of floating point (FP) operations, but at the expense of increasing the number of conditional branches and compare instructions.

We used a refactored version of the *GMM* function to remove all the conditional branches that are due to the selection of top 4 Gaussians. In this version each iteration is simpler as we remove the conditional branch and compare instructions that are required to check whether the Gaussian must be discarded. We refer to this version as *All32*, as the 32 Gaussians in the mixture are always used to compute senone scores.

Listing 2 shows the x86 assembly code for the innermost loop of both implementations, the baseline version using the top 4 Gaussians and *All32*. Top 4 requires 4 FP operations (2 subtractions and 2 multiplications) per loop

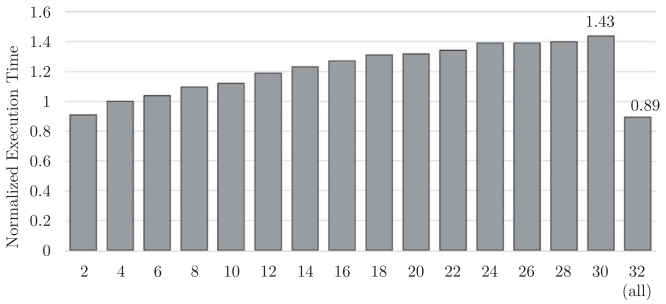


Fig. 7. Normalized execution time for different *top N* Gaussians with respect to *top 4*.

iteration. Furthermore, 2 conditional branches are included, one for branching at the beginning of the loop and one for branching outside the loop. As 28.6 iterations are performed on average, this version executes 114.44 FP operations and 57.22 conditional branches per senone.

On the other hand, *All32* version requires the same 4 FP operations, but only one conditional branch is executed per loop iteration. Since 36 iterations are performed per senone, *All32* executes 144 FP operations and 36 conditional branches. Therefore, this implementation increases FP operations by 25.8 percent, but it reduces conditional branches by 37 percent. We found that *All32* outperforms the version that selects top 4 Gaussians, as reported in Section 6.

Using the top N Gaussians was proposed as an optimization [13] in 2001, targeting significantly different pipelines for which FP operations were more expensive and branches were less costly than in today's microprocessors. Our experimental results show that this technique is not beneficial for modern CPUs, as these CPUs excel in FP performance but conditional branches are one of the main sources of stalls.

Fig. 7 shows the normalized execution time for selecting different top Gaussians with respect to the baseline (top 4). As the figure shows, selecting more number of top Gaussians results in significant increase in the execution time. While selecting the *top N* Gaussians seems to reduce the floating-point operations, an increase in the number of branches results in more stalls and performance degradation.

4.2 Vectorization

GMM evaluation code is a good candidate for vectorization as the innermost loop iterations are independent. However, each iteration only includes four FP operations that must be executed sequentially and, hence, they cannot be packed in the same SIMD instruction. Due to this lack of independent FP operations, the innermost loop cannot be efficiently vectorized as it is. One effective way of exposing more independent FP instructions is using loop unrolling. We use unrolling factor equal to the SIMD width, i.e., vector size. In addition, we exploit FMA instructions to merge the last two FP operations performed in the scalar loop, in order to further reduce instruction count.

4.3 Improved Memory Layout

In the SIMD version, multiple components of a Gaussian are computed at the same time by using multiple SIMD lanes. For a vector size of 4, each one of the 4 SIMD lanes computes and accumulates the values for 9 components. In order to get the acoustic likelihood, the aggregated value for the 36

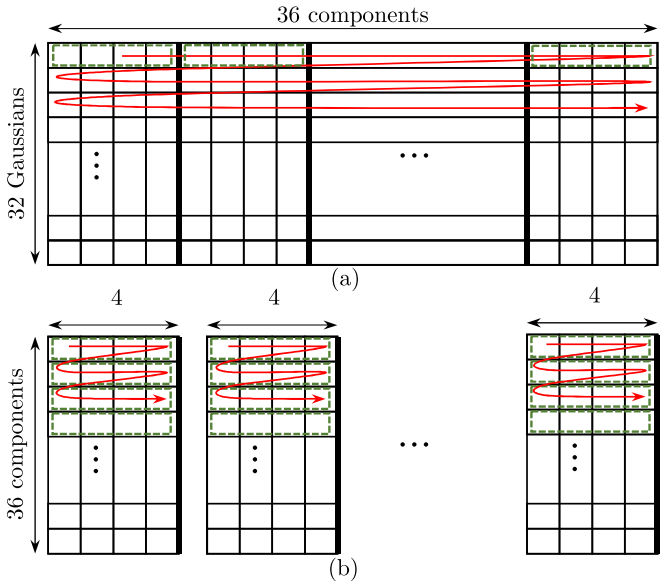


Fig. 8. Memory layout and memory access pattern. (a) The baseline and (b) the transposed layout.

components has to be computed, so we have to add the 4 partial results obtained by the 4 SIMD lanes. This reduction is performed using scalar instructions.

In an attempt to maximize the amount of vectorizable code, we propose to use the SIMD lanes to compute multiple Gaussians at the same time, instead of processing multiple components of one Gaussian. By doing this, the accumulated value on each SIMD lane is the final likelihood for one Gaussian and, hence, no horizontal reduction is required. This implementation requires some changes to the memory layout of *means* and *vars* matrices, as it is not possible to fetch the same component (or column) for different Gaussians (or rows) with one vector load, as they are not stored consecutively in memory.

We propose to change the memory layout for these matrices to the one illustrated in Fig. 8b. Transposing the matrix solves the problem with the vector load, as now we fetch consecutive columns (or Gaussians) in the same row (or component). However, by transposing the matrix we lose some spatial locality, as the traversal is performed in column-major order. Note that each SIMD lane has to process and accumulate results for one column. Our solution to this problem is to split the matrix in smaller sub-matrices with a number of columns equal to the SIMD width. By doing this we store the data in memory in the same order it is accessed by the application, maximizing spatial locality.

The proposed memory layout eliminates the horizontal reduction, increasing the amount of vectorizable code. Therefore, we reduce the number of scalar instructions to a large extent, achieving significant performance and energy savings over the first SIMD version as shown in Section 6. On the other hand, we found that GMM evaluation benefits from larger SIMD widths, achieving better results when using AVX (SIMD width of 8) than SSE (SIMD width of 4).

4.4 Multi-Frame Gaussian Evaluation

The power/performance analysis of Pocketsphinx, presented in Section 3.2, identified system memory as the main source of both CPU stalls and energy drain. The SIMD

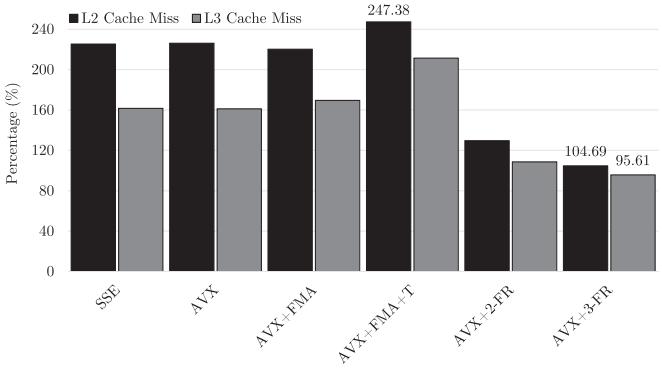


Fig. 9. Number of L2 and L3 cache misses normalized to the baseline.

implementation presented in Section 4.3 exacerbates the problem due to memory latency. Due to the higher throughput of the VPU, the pressure on the memory subsystem increases because data is requested earlier than in the scalar implementation. Therefore, the prefetcher has less time to bring the data from memory and it is not able to achieve timeliness, which leads to an increase in the number of L2 and L3 misses, as shown in Fig. 9.

As we describe in Section 3, most of the memory bandwidth is used to fetch Gaussian parameters, i.e., means and variances. Those accesses exhibit poor temporal locality due to the big size of the dataset for one frame of speech. CPU caches cannot exploit frame-to-frame reuse and, hence, Gaussian parameters have to be fetched from system memory on a frame basis.

One approach to reduce the number of accesses to system memory is to evaluate Gaussians for multiple frames at the same time. By using multi-framing, means and variances for one Gaussian are fetched once in the on-chip caches and are used to evaluate the Gaussian in several frames of speech. For example, if we merge Gaussian evaluation for two frames, then the bandwidth usage is reduced by approximately one half, as means and variances are fetched from memory every other frame. Moreover, the number of FP operations per memory access doubles, alleviating the pressure on the memory subsystem and especially on the prefetcher.

The main hurdle for implementing the multi-frame approach is the lack of information about active senones for subsequent frames. The list of active senones is generated by the search after the pruning and it is only available for the current frame. We could decouple GMM evaluation from the search by ignoring the list of active senones and computing GMM for all of them. By doing this we could implement the multi-frame approach, but the workload would increase by 2x, as only half of the senones are active on average. Such a huge overhead renders this naive approach completely ineffective.

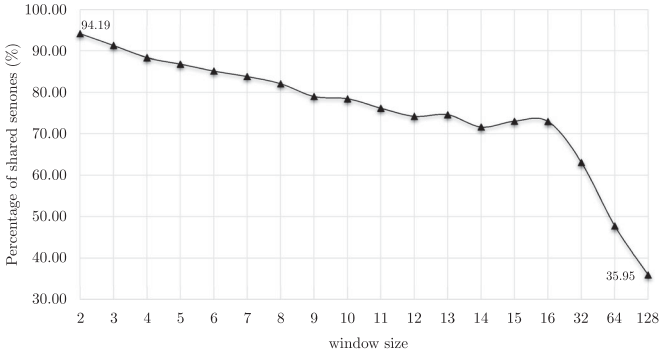


Fig. 10. Percentage of shared active senones among consecutive frames for different window sizes.

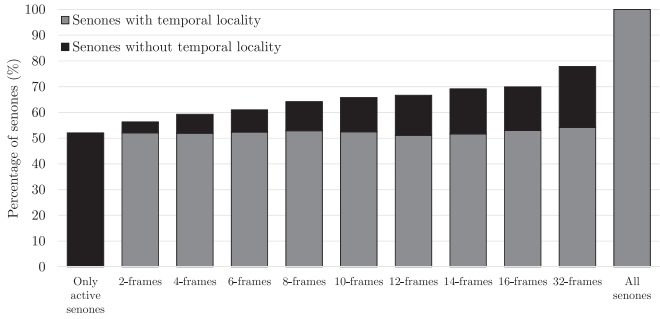


Fig. 11. Gray bars show the percentage of senones computed with the multi-frame version, i.e., percentage of senones computed exploiting temporal locality. Black bars show the percentage of senones computed with the original single-frame code, i.e., the percentage of senones for which temporal locality is not exploited. For the versions computing 2-32 frames at a time, the black bars correspond to the mispredicted senones.

An analysis of the locality of active senones reveals a better strategy. The speech signal is quasi-stationary when considering small intervals, so the list of active senones tend to be similar for consecutive frames. Fig. 10 shows the percentage of active senones that are shared among consecutive frames, for different window sizes. As we can see, considering a window of 2-3 frames more than 90 percent of the active senones are shared among those frames.

We use this observation to implement a simple prediction scheme. Our scheme predicts that the active senones for the next N - 1 frames will be the same as those in the current frame. GMM computation is triggered just once every N frames, using the multi-framing approach of fetching parameters once and computing the Gaussians for N frames. In addition, we include a recovery mechanism to handle mispredictions. Two types of mispredictions are possible. The first type happens when we predict a senone as active but it is inactive. In this case we do not have to trigger any action, but we pay the overhead of computing an acoustic score that is not used during the search process. The second type happens when we predict a senone as inactive but it is active. In this case we trigger the single-frame version of Gaussian Evaluation to compute the score for the mispredicted senone. Fig. 11 shows that this scheme is very effective as the number of mispredictions is very small (mispredicted senones correspond to the black bars, whereas correctly predicted senones are shown in gray bars), so we can exploit temporal locality for most of the senones. As we can see in both Figs. 10 and 11, increasing the number of frames computed at a time, i.e., the window size, increases the number of mispredicted senones, as the speech signal changes significantly at long distances. We obtained the best performance and power results by using small windows of 2-3 frames, since bigger windows suffer the overhead of mispredicted senones and provide diminishing returns in memory bandwidth savings.

TABLE 2
Intel Haswell and Skylake Parameters

	Haswell	Skylake
Core	Intel i5-4210M	Intel i7-6700
L1, L2, L3	32 KB, 256 KB, 3 MB	64 KB, 256 KB, 8 MB
Frequency	3.2 GHz	4.2 GHz
Main Memory	8 GB DDR3	32 GB DDR4

5 Evaluation Methodology

We have evaluated our techniques using two high-end Intel desktop CPUs, whose parameters are shown in Table 2. We use PAPI [14] hardware performance counters on Haswell and Skylake processors to measure execution time, whereas we employ Intel RAPL [15] library to collect energy consumption. Furthermore, we have evaluated our optimizations on a low-power mobile ARM A57 CPU with parameters shown in Table 3. We use the NVIDIA Tegra X1 [16] SoC to collect execution time. To measure energy consumption, we read the registers of the TI INA3221 power monitor included in the NVIDIA Jetson TX1 platform, in order to obtain power dissipation by monitoring CPU power rail as described in [17].

On the other hand, we use Sniper simulator [18] to collect further information of the CPU pipeline, including a complete CPI stack. We model a modern out-of-order mobile CPU, similar to an Atom Bay Trail processor [19]. The parameters for the experiments are included in Table 1. We use McPAT [20] to estimate energy consumption of the Atom-like processor. We use GCC version 4.8 in the $\times$86 and ARM platforms and we employ -O3 optimization level.

We use standard audio files commonly employed to test ASR systems as our datasets. More specifically, we use LibriSpeech [21] corpus including 5.4 hours of audio files.

Our baseline for the experiments is the unmodified Pocketsphinx GMM implementation. We have evaluated the effect of our techniques on five different acoustic models trained in Sphinx. The parameters of the different acoustic models are shown in Table 4.

6 Experimental Results

In this section, we analyze the performance and energy efficiency of the optimizations presented in Section 4. The baseline configuration for all our experiments is the unmodified Pocketsphinx (Section 3). Fig. 12 shows the speedup and normalized energy achieved by all the optimizations on an Intel Haswell CPU, using the English1 acoustic model with parameters shown in Table 4. Note that we build each optimization on top of the previous one and the performance and energy improvements are measured for the entire application. The speedups and energy savings are reported for the entire ASR pipeline required to convert the speech into words, including the Feature Extraction and Viterbi Search in addition to the GMM evaluation. *All32* configuration improves performance by 12.8 percent and saves 11.2 percent energy by removing conditional branches in the innermost loop of GMM evaluation. *All32* reduces conditional branches by 37 percent with respect to the baseline, at the cost of increasing FP operations by 25 percent. Our results show that this is a good trade-off for modern CPUs, as the penalties introduced by branches are bigger than the cost of the extra FP operations. The results for the straightforward SIMD implementation, introduced in Section 4.2, are included for *SSE* and *AVX*. *SSE* employs a SIMD width of 4 and achieves 72.8 percent speedup and 34.4 percent energy savings. *AVX* version slightly improves the results to 76.9 percent speedup and 40.8 percent energy savings by using a SIMD width of 8. The use of Fused Multiply-Add (FMA) instruction further improves speedup to 78.8 percent and energy savings to 47.2 percent as shown in configuration *AVX+FMA*. The speedups of the SIMD version come from the higher FP throughput of the VPU. The energy savings come from the smaller execution time (static energy) and the reduction in instruction count (dynamic energy), as multiple scalar operations are packed in just one SIMD instruction.

Configuration *AVX+FMA+T* implements the improved memory layout presented in Section 4.3. By using this layout for matrices that store means and variances the amount of vectorizable code increases, further improving speedup to 96 percent and energy savings to 47.5 percent.

Finally, configurations *AVX+2FR* and *AVX+3FR* implement the multi-framing scheme presented in Section 4.4 using a window of 2 and 3 frames respectively to compute the Gaussians. *AVX+3FR* achieves the best results, providing 2.63x speedup and 61 percent energy savings. This

TABLE 3
Mobile CPU Parameters

CPU	4x ARM Cortex A57
L1, L2	32 KB L1, 2 MB L2
Technology	20 nm
Frequency	1.9 GHz
Main Memory	4 GB LPDDR3

TABLE 4
Parameters for the Different Acoustic Models That Are Employed to Evaluate Our Proposed Techniques

Acoustic Model	#Mixtures	#Gaussians	Dimension
English 1	5,138	32	36
English 2	6,126	32	39
German	6,198	16	29
Russian	5,147	32	36
Greek	5,102	32	36

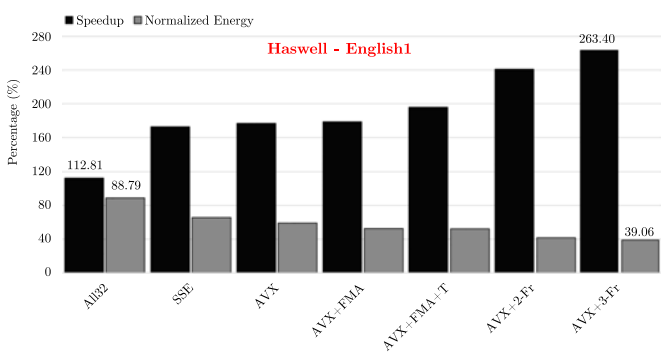


Fig. 12. Speedup and normalized energy on Intel Haswell CPU. Baseline is unmodified Pocketsphinx with the English1 acoustic model.

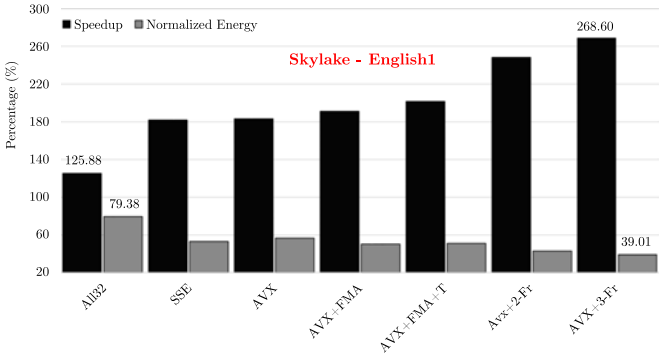


Fig. 13. Speedup and normalized energy on Intel Skylake CPU. Baseline is unmodified Pocketsphinx with the English1 acoustic model.

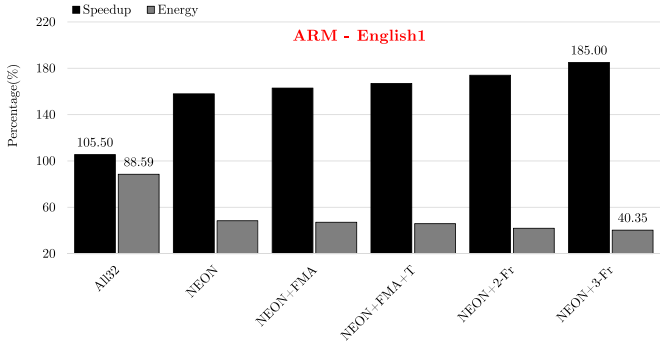


Fig. 15. Speedup and normalized energy on ARM Cortex-A57 mobile CPU. Baseline is unmodified Pocketsphinx with the English1 acoustic model.

multi-framing scheme reduces the number of accesses to system memory as means and variances are fetched every 3 frames instead of being fetched on a frame basis. This reduces DRAM energy and also improves performance by alleviating the pressure on the memory subsystem, as memory latency is the main performance limiting factor (see Section 3.2).

The multi-framing approach uses a prediction schemes that assumes that the active senones do not change for a group of N consecutive frames. This prediction is very effective for small sizes of N, such as 2 or 3, as the speech signal is quasi-stationary when considering small intervals. We found that 254 senones are mispredicted on average per frame out of 2,929 senones computed, so misprediction rate is only 8.6 percent. In the single-frame version 9,344 bytes/senone are fetched from system memory: 128 bytes for determinant array, 4,608 bytes for means matrix and 4,608 for variance matrix. Since 2,675 senones are active on average, 23.83 MBytes are fetched from system memory per frame. With *AVX+3FR* configuration, determinants, means and variances are fetched from system memory every 3 frames. Hence, the amount of data accessed per frame is reduced to $23.83/3 = 7.94$ MBytes. For multi-framing, we also have to consider memory accesses to compute mispredicted senones: 254 mispredicted senones/frame $\times$ 9344 bytes/senone = 2.26 MBytes. So the total amount of data fetched from memory per frame with *AVX+3FR* is 10.2 MBytes, a reduction of 57.1 percent with respect to the single-frame version.

We tested the multi-framing approach using bigger numbers of frames, from 4 to 16. However, we obtained better results for small windows of just 2-3 frames for several

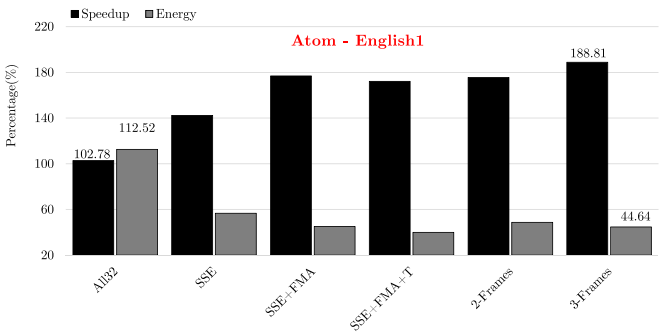


Fig. 14. Speedup and normalized energy on Intel Atom CPU. Baseline is unmodified Pocketsphinx with the English1 acoustic model.

reasons. First, as we increase the number of frames we get diminishing returns in bandwidth savings and, therefore, in speedups and energy savings. Second, the number of mispredicted senones increases for bigger windows of frames (see Fig. 10), increasing the overheads. Third, all the data for computing a Gaussian in multiple frames can be stored in the vector register file for small windows, but for a big number of frames L1 must be used, increasing memory pressure.

The proposed optimizations achieve similar speedups and energy savings when they are applied on an Intel Skylake CPU (Intel latest microprocessor), as illustrated in Fig. 13. The best configuration, *AVX+3FR*, provides 2.68x speedup and 61 percent energy savings.

We have also evaluated our optimizations on an Atom-like mobile CPU by running simulations on Sniper. The performance and energy results are shown in Fig. 14. Note that Atom does not support AVX. As in Skylake and Haswell, on Atom the multi-frame implementation with window size of 3 frames achieves the best results, providing 1.88x speedup and 55.3 percent energy savings. As Fig. 14 shows, a slight increase can be seen in the energy consumption of *All32*. This outlier is indeed due to the McPAT power model. McPAT accurately accounts for the energy of the extra FP operations and cache accesses in this configuration. However, it does not properly model the cost of a recovery from a branch misprediction. Energy for flushing the pipeline or recovering the register map table is not considered in McPAT.

Since one of the main targets of Pocketsphinx are mobile devices, we have also evaluated our optimizations on a state-of-the-art ARM mobile CPU. The performance and energy results are shown in Fig. 15. ARM CPUs take advantage of NEON extensions, which provide vector instructions with SIMD width of 4. Removing conditional branches provides 5.5 percent performance improvement while reducing energy consumption to 11 percent. Similar to the Intel desktop CPUs, removing conditional branches at the cost of increasing FP operations is also a good trade-off in the ARM CPU. We get the best results with the multi-frame implementation with window size of 3 frames, that provides 1.85x speedup and 59.65 percent energy savings.

Regarding the use of desktop processors in the evaluations, we consider that it is also important to optimize ASR for high-end processors. Energy consumption is also an

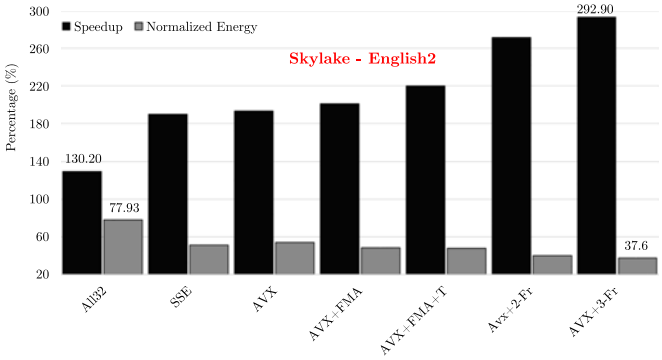


Fig. 16. Speedup and normalized energy on Intel Skylake CPU, using the English2 acoustic model. Baseline is unmodified Pocketsphinx.

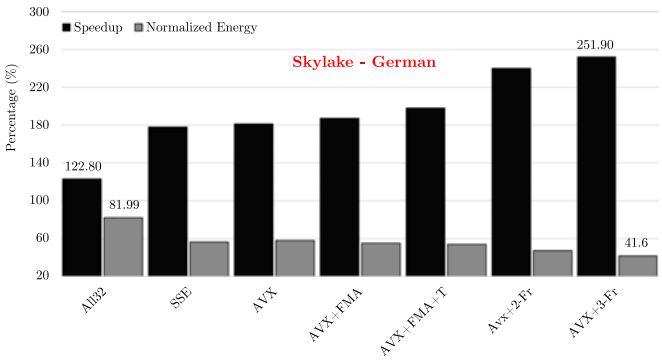


Fig. 17. Speedup and normalized energy on Intel Skylake CPU, using the German acoustic model. Baseline is unmodified Pocketsphinx.

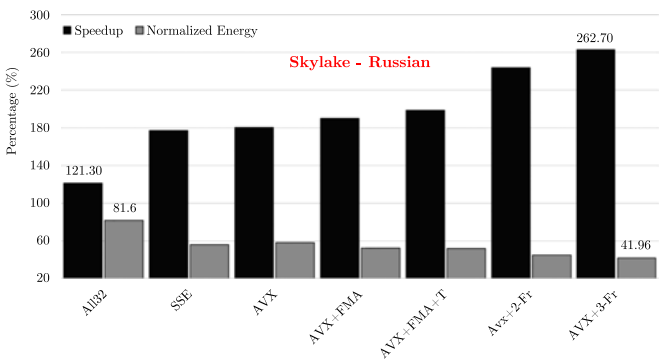


Fig. 18. Speedup and normalized energy on Intel Skylake CPU, using the Russian acoustic model. Baseline is unmodified Pocketsphinx.

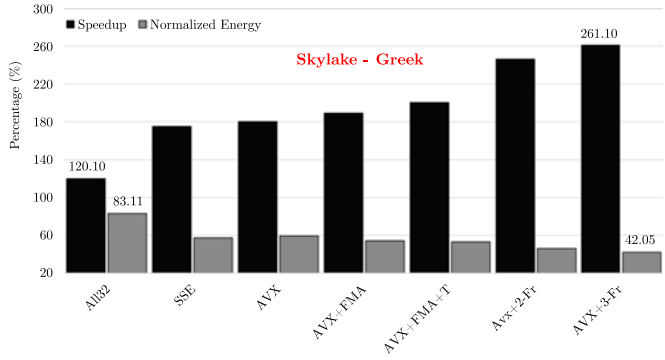


Fig. 19. Speedup and normalized energy on Intel Skylake CPU, using the Greek acoustic model. Baseline is unmodified Pocketsphinx.

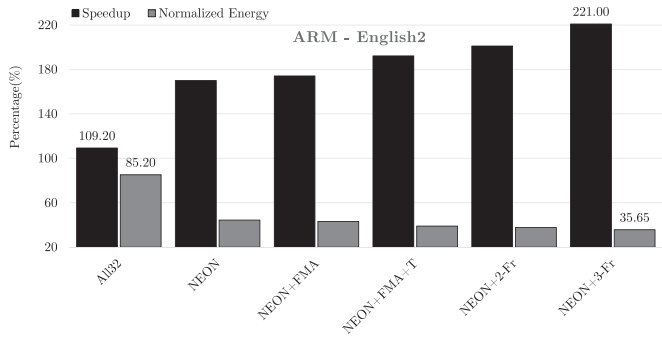


Fig. 20. Speedup and normalized energy on ARM Cortex-A57 mobile CPU, using the English2 acoustic model. Baseline is unmodified Pocketsphinx.

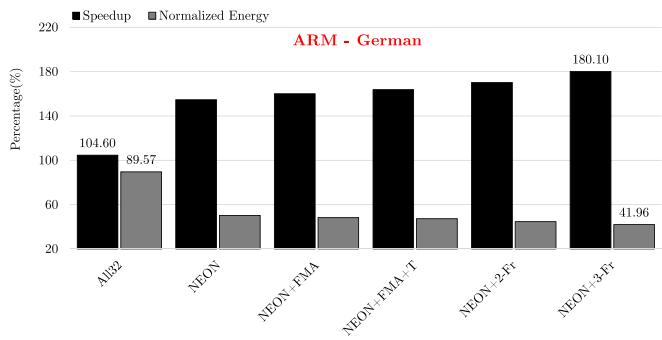


Fig. 21. Speedup and normalized energy on ARM Cortex-A57 mobile CPU, using the German acoustic model. Baseline is unmodified Pocketsphinx.

important issue for desktops, due to heat dissipation, and for servers, as it affects the cost of operating data centers.

To illustrate the general applicability of our techniques, we have evaluated the speedups and energy savings for different acoustic models. Figs. 16, 17, 18, and 19 show the results on the Intel Skylake CPU for the English2, German, Russian and Greek acoustic models respectively, whose parameters are shown in Table 4. As it can be seen, our techniques provide substantial speedups and energy savings for acoustic models with different parameters that target different languages. The configuration *AVX+3FR* achieves the best results, the speedups for the different acoustic models range between 2.51x (German) and 2.92x (English2). Regarding the energy savings, *AVX+3FR* reduces energy by 62.4 and 58.4 percent for English2 and German, respectively.

Finally, Figs. 20, 21, 22, and 23 show the speedups and energy savings on the ARM mobile CPU for the English2, German, Russian and Greek acoustic models respectively. The results are similar to the benefits reported for English1 in Fig. 15: our techniques provide consistent performance improvements and energy savings for different acoustic models.

6.1 Comparison with Other GMM Implementations

GMM is a machine learning technique used in a wide range of applications in different areas including, for example, speech recognition or image recognition. A popular implementation of GMM evaluation consists on using matrix-matrix multiplication as described in [22]. In this implementation, the Gaussians and the input features are represented as 2D matrices and the acoustic scores are obtained by

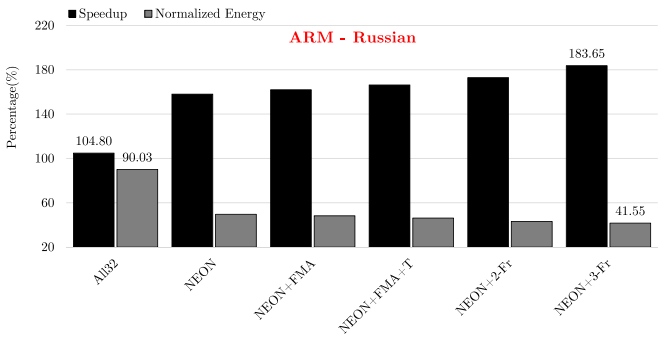


Fig. 22. Speedup and normalized energy on ARM Cortex-A57 mobile CPU, using the Russian acoustic model. Baseline is unmodified Pocketsphinx.

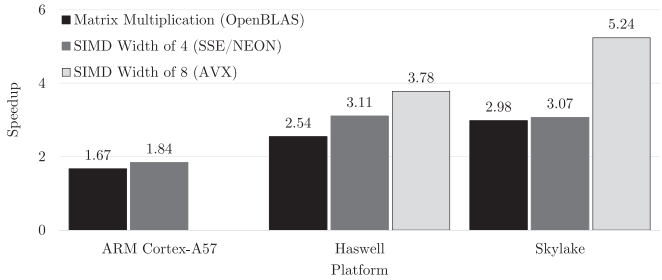


Fig. 24. Speedup achieved by matrix multiplication technique using OpenBLAS library versus our proposed techniques running on different platforms. Baseline is unmodified Pocketsphinx. All the configurations implement the same acoustic model for English language.

performing matrix multiplication, typically by using the BLAS specification for dense linear algebra. This is the approach employed in other speech recognition toolkits like Kaldi [23].

In this section, we compare the performance and energy consumption of our GMM implementation with the matrix multiplication approach. We use OpenBLAS library [24], a high-performance implementation of the BLAS specification, to implement Pocketsphinx's acoustic model by leveraging the high-performance implementation of the SGEMM operation (single-precision general matrix multiplication). In our experiments, we use single-threaded OpenBLAS implementation. Fig. 24 shows the speedups of different GMM implementations of the last acoustic model for English language in Pocketsphinx. Baseline is the unmodified Pocketsphinx implementation. Our version of GMM outperforms the matrix multiplication approach in the Intel and ARM platforms. We achieve 10, 22 and 3 percent speedup when running on ARM, Haswell and Skylake CPUs respectively, when using a SIMD width of 4 (same than OpenBLAS). On the other hand, Fig. 25 shows the normalized energy for the same configurations and CPUs. Our GMM implementation provides higher energy savings than the version based on matrix multiplication.

Our optimized decoder achieves higher performance and energy-efficiency than the GMM implementation based on matrix multiplication due to several reasons. First, instruction mix analysis of the different implementations reveals that 45 percent of the instructions in our proposed methods are SIMD instructions, whereas 38 percent of the instructions in matrix multiplication are vector instructions. Second, matrix multiplication implementation requires a preprocessing stage to prepare the matrix of input features. Although the matrix with the Gaussians, i.e., means and variances, is static and can be initialized offline, the matrix with the input features has to be created on-the-fly for each frame of speech as described in [22]. This preprocessing represents a non-negligible overhead. In comparison with the matrix multiplication, our *SSE+FMA+T* implementation requires 13 percent less instructions.

6.2 Discussion

Our proposed methods are applicable for any acoustic model based on Gaussian Mixture Models, so it works for speech in any language. In this paper, we use Pocketsphinx as our baseline ASR system since we target mobile platforms, but our proposed techniques can be used in the acoustic models available in Sphinx 4, Kaldi, Julius or HTK. We believe speech recognition will be a feature supported by the majority of computing devices in the near future, and acoustic models will evolve towards more complex ones for the sake of better accuracy.

On the other hand, the proposed techniques can also be used for other applications that are relevant for mobile devices, especially in the area of computer vision. GMMs are employed for image segmentation [25], [26], image retrieval [27], tracking people in images [28] or detecting and tracking moving objects in video sequences [29].

7 Related Work

Improving performance of GMM computation for speech recognition has attracted the attention of the research community the last few years. Regarding software improvements, most proposals focus on reducing the amount of computation at the cost of increasing Word Error Rate. In PTM acoustic model [5], [30] triphones that belong to the

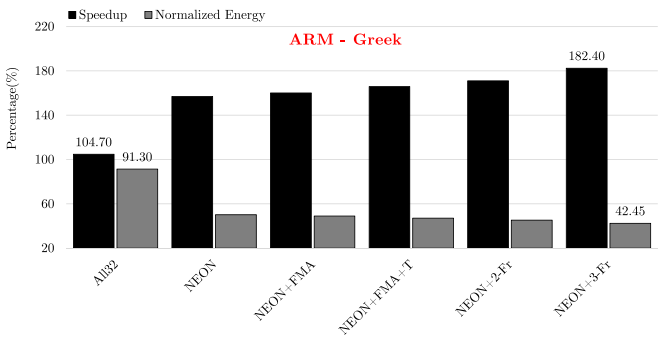


Fig. 23. Speedup and normalized energy on ARM Cortex-A57 mobile CPU, using the Greek acoustic model. Baseline is unmodified Pocketsphinx.

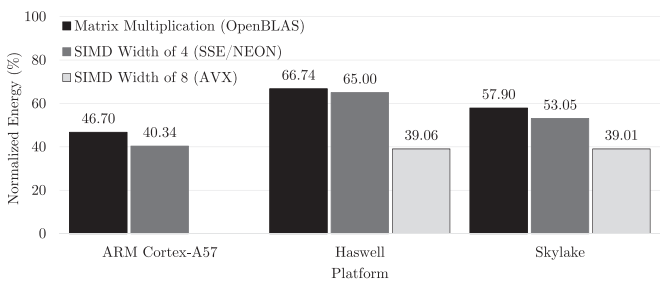


Fig. 25. Normalized energy consumption achieved by matrix multiplication technique using OpenBLAS library versus our proposed techniques running on different platforms. Baseline is unmodified Pocketsphinx.

same basephone share one GMM, reducing the number of Gaussians evaluated for acoustic scoring in two orders of magnitude. Partial Distance Elimination [13], [31] employs the N Gaussians with highest likelihood to compute senone score instead of using all the Gaussians. Our work is different as we maintain accuracy and boost performance by exploiting the VPU and saving bandwidth with the multi-frame approach.

The use of SIMD execution model for GMM computation has been subject of research for several years [32], [33], [34]. In contrast to our work, these proposals do not evaluate the impact of SIMD on energy consumption. Moreover, we evaluate the interaction of SIMD instructions with hardware prefetchers, and we change the memory layout to maximize the amount of vectorizable code.

Gupta et al. [11], [35] propose a chunk-based technique to compute GMMs that is similar to our multi-framing approach. Our work is different in several ways. First, we target CPUs instead of GPUs. Second, our baseline employs much bigger vocabulary (130 k words versus 5 k) and acoustic model (164 k Gaussians versus 15 k), so our datasets significantly exceed the capacity of the on-chip caches. Finally, we introduce a recovery mechanism to handle mispredictions.

Tan et al. [36] explain in detail how Automatic Speech Recognition can be presented in mobile devices and also over communication networks. In case the ASR is provided in the network, the voice is captured in user's device while it is processed in the cloud. Although complex and more accurate ASR systems can be executed in the cloud, for many tasks it is prefered to provide ASR in user's device due to indeterminate response-time in the cloud, inaccessibility to the network or security reasons.

Previous work proposed the use of GPUs [37], [38], [39], [40] or dedicated accelerators [10], [41], [42] for GMM computation, achieving substantial speedups. However, GPUs exhibit high power consumption and big area, so they are not amenable for small ultra low power devices. The main concerns with dedicated accelerators are the lack of flexibility and a long development cycle. In our research we target real-time software-based ASR systems running on CPUs.

7.1 Deep Neural Networks for Acoustic Scoring

GMM has been the mainstream machine learning technique for implementing the acoustic model in speech recognition systems for decades. The vast majority of ASR systems, such as Sphinx, Pocketsphinx, Kaldi or HTK, provide an implementation of acoustic scoring based on GMMs. In recent years, the use of Deep Neural Networks (DNNs) for acoustic scoring [43] has become very popular due to its high recognition accuracy. Unlike the conventional idea that these are two competing approaches for speech recognition, recent research has shown that GMMs and DNNs complement each other. Swietojanski et al. [44] propose an acoustic model that combines a GMM and a DNN, achieving higher accuracy than the DNN alone. Rath et al. [45] present a hybrid and stacked ASR system that also combines a DNN with a GMM to improve recognition accuracy. These ASR systems combine the frame-level acoustic scores computed separately by a DNN and a GMM. Other hybrid systems, like the tandem approach [46] or the bottleneck approach [47], employ DNNs as feature extractors for a GMM. Yu et al. [48] explain in detail fuse DNN and GMM systems. Recently, Tachioka et al. [49] proposed to use DNN and GMM-based ASR systems to address variety of noises in a noisy environment. Their results show that combining these approaches increases the accuracy of the ASR system significantly.

To sum up, GMMs are still the most common technique for implementing the acoustic scoring in ASR systems. Furthermore, previous work has shown the synergy between GMM and DNN techniques. Therefore, we believe that improving the performance and energy-efficiency of GMMs will be of special interest for future speech recognition systems.

8 Conclusion

In this paper we present an energy/performance analysis of Automatic Speech Recognition system when running on a general purpose CPU. We show that the Gaussian evaluation of the acoustic model is the most computationally expensive component, as it represents 81.3 percent of total execution time. Most of the CPU stalls are due to mispredicted branches and accesses to system memory. Regarding energy consumption, DRAM is clearly the main source of energy drain.

We propose multiple optimizations to alleviate the bottlenecks identified in the analysis. First, we remove conditional branches from the innermost loop of the Gaussian evaluation code, achieving 12 percent speedup and 11 percent energy savings. Second, we employ a multi-frame Gaussian evaluation scheme with prediction of active senones to reduce off-chip memory accesses by 57.1 percent.

Finally, we exploit the VPU via SIMD instruction and a new memory layout to boost Gaussian evaluation and improve energy efficiency. Our implementation using SIMD instructions and multi-frame Gaussian evaluation achieves 2.68x speedup and 61 percent energy savings on an Intel Skylake CPU. Furthermore, it obtains 1.88x and 1.85x speedup and reduces energy consumption by 55 and 59 percent on an Atom-like and a modern ARM mobile CPUs respectively. The performance improvements and energy savings are achieved without any loss in the accuracy of the ASR system.

Acknowledgments

This work was supported by the Spanish State Research Agency under grants TIN2013-44375-R and TIN2016-75344-R (AEI/FEDER, EU).

References

[1] Siri. [Online]. Available: https://en.wikipedia.org/wiki/Siri
[2] Cortana. [Online]. Available: https://en.wikipedia.org/wiki/Cortana__%28software%29
[3] Google Now. [Online]. Available: https://en.wikipedia.org/wiki/Google_Now
[4] D. Huggins-Daines, M. Kumar, A. Chan, A. Black, M. Ravishankar, and A. Rudnicky, "PocketSphinx: A free, real-time continuous speech recognition system for hand-held devices," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, May 2006, pp. I-185–I-188.
[5] A. Lee, T. Kawahara, K. Takeda, and K. Shikano, "A new phonetic tied-mixture model for efficient decoding," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, 2000, pp. 1269–1272.

[6] C. Gaida, P. Lange, R. Petrick, P. Proba, A. Malatawy, and D. Suendermann-Oeft, "Comparing open-source speech recognition toolkits," Project OASIS DHBW Stuttgart, Stuttgart, Germany, 2014.

[7] L. Rabiner, "A tutorial on hidden Markov models and selected applications in speech recognition," *Proc. IEEE*, vol. 77, no. 2, pp. 257–286, Feb. 1989.

[8] R. Yazdani, A. Segura, J.-M. Arnau, and A. Gonzalez, "An ultra low-power hardware accelerator for automatic speech recognition," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2016, pp. 1–12.

[9] R. Yazdani, A. Segura, J.-M. Arnau, and A. Gonzalez, "Low-power automatic speech recognition through a mobile GPU and a Viterbi accelerator," *IEEE Micro*, vol. 37, no. 1, pp. 22–29, Jan./Feb. 2017.

[10] B. Mathew, A. Davis, and Z. Fang, "A low-power accelerator for the SPHINX 3 speech recognition system," in *Proc. Int. Conf. Compilers Archit. Synthesis Embedded Syst.*, 2003, pp. 210–219. [Online]. Available: http://doi.acm.org/10.1145/951710.951739

[11] K. Gupta and J. D. Owens, "Three-layer optimizations for fast GMM computations on GPU-like parallel processors," in *Proc. IEEE Workshop Automatic Speech Recognit. Understanding*, 2009, pp. 146–151.

[12] L. Bahl, et al., "Further results on the recognition of a continuously read natural corpus," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, Apr. 1980, pp. 872–875.

[13] B. L. Pellom, R. Sarikaya, and J. H. Hansen, "Fast likelihood computation techniques in nearest-neighbor based search for continuous speech recognition," *IEEE Signal Process. Lett.*, vol. 8, no. 8, pp. 221–224, Aug. 2001.

[14] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "PAPI: A portable interface to hardware performance counters," in *Proc. Dept. Defense HPCMP Users Group Conf.*, 1999, pp. 7–10.

[15] V. Weaver, et al., "Measuring energy and power with PAPI," in *Proc. 41st Int. Conf. Parallel Process. Workshops*, 2012, pp. 262–268.

[16] NVIDIA Tegra X1. [Online]. Available: http://international.download.nvidia.com/pdf/tegra/Tegra-X1-whitepaper-v1.0.pdf

[17] Merlin Friesen, "Linux power management optimization on the Nvidia Jetson platform," [Online]. Available: http://events.linuxfoundation.org/sites/events/files/slides/Linux_Low_Power_ELC_SanDiego.pdf

[18] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Trans. Archit. Code Optimization*, vol. 11, 2014, Art. no. 28.

[19] Intel Atom Processor (Bay Trail). [Online]. Available: http://ark.intel.com/products/84311/Intel-Atom-Processor-E3805-1M-Cache-1_33-GHz

[20] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2009, pp. 469–480.

[21] V. Panayotov, G. Chen, D. Povey, and S. Khudanpur, "Librispeech: An ASR corpus based on public domain audio books," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, 2015, pp. 5206–5210.

[22] P. R. Dixon, T. Oonishi, and S. Furui, "Harnessing graphics processors for the fast computation of acoustic likelihoods in speech recognition," *Comput. Speech Language*, vol. 23, no. 4, pp. 510–526, 2009.

[23] D. Povey, et al., "The kaldi speech recognition toolkit," in *Proc. IEEE Workshop Automatic Speech Recognit. Understanding*, 2011.

[24] An optimized BLAS library (OpenBLAS). [Online]. Available: http://www.openblas.net/

[25] R. Farnoosh and B. Zarpak, "Image segmentation using gaussian mixture model," *IUST Int. J. Eng. Sci.*, vol. 19, no. 1/2, pp. 29–32, 2008.

[26] N. Greggio, A. Bernardino, C. Laschi, P. Dario, and J. Santos-Victor, "Fast estimation of gaussian mixture models for image segmentation," *Mach. Vis. Appl.*, vol. 23, no. 4, pp. 773–789, 2012.

[27] Z. Robotka and A. Zempléni, "Image retrieval using Gaussian mixture models," *Ann. Univ. Sci. Budapest Sect. Comp*, vol. 31, pp. 93–105, 2009.

[28] G. Moradiannejad, "People tracking under occlusion using Gaussian mixture model and fast level set energy minimization," Ph.D. dissertation, Dept. Elect. Comput. Eng., Univ. Ottawa, Ottawa, ON, USA, 2013.

[29] P. Chockalingam, "Non-rigid multi-modal object tracking using Gaussian mixture models," Ph.D. dissertation, Graduate School, Clemson Univ., Clemson, SC, USA, 2009.

[30] A. Sankar, "A new look at HMM parameter tying for large vocabulary speech recognition," in *Proc. Int. Conf. Spoken Languagessss Process.*, 1998, pp. 2219–2222.

[31] J. Cai, G. Bouselmi, Y. Laprie, and J.-P. Haton, "Efficient likelihood evaluation and dynamic gaussian selection for HMM-based speech recognition," *Comput. Speech Language*, vol. 23, no. 2, pp. 147–164, 2009.

[32] S. Kanthak, K. Schutz, and H. Ney, "Using SIMD instructions for fast likelihood calculation in LVCSR," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, 2000, pp. 1531–1534.

[33] J. Ou, J. Cai, and Q. Lin, "Using SIMD technology to speed up likelihood computation in HMM-based speech recognition systems," in *Proc. Int. Conf. Audio Language Image Process.*, 2008, pp. 123–127.

[34] P. Cardinal, P. Dumouchel, and G. Boulianne, "Large vocabulary speech recognition on parallel architectures," *IEEE Trans. Audio Speech Language Process.*, vol. 21, no. 11, pp. 2290–2300, Nov. 2013.

[35] K. Gupta and J. D. Owens, "Compute & memory optimizations for high-quality speech recognition on low-end GPU processors," in *Proc. 18th Int. Conf. High Performance Comput.*, 2011, pp. 1–10.

[36] Z.-H. Tan and B. Lindberg, *Automatic Speech Recognition on Mobile Devices and Over Communication Networks*. Berlin, Germany: Springer, 2008.

[37] J. Vaněk, J. Trmal, J. V. Psutka, and J. Psutka, "Optimized acoustic likelihoods computation for NVIDIA and ATI/AMD graphics processors," *IEEE Trans. Audio Speech Language Process.*, vol. 20, no. 6, pp. 1818–1828, Aug. 2012.

[38] P. Cardinal, P. Dumouchel, G. Boulianne, and M. Comeau, "GPU accelerated acoustic likelihood computations," in *Proc. Annu. Conf. Int. Speech Commun. Assoc.*, 2008, pp. 964–967.

[39] K. You, et al., "Parallel scalability in speech recognition," *IEEE Signal Process. Mag.*, vol. 26, no. 6, pp. 124–135, Nov. 2009.

[40] P. R. Dixon, T. Oonishi, and S. Furui, "Fast acoustic computations using graphics processors," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, 2009, pp. 4321–4324.

[41] B. Mathew, A. Davis, and M. Parker, "A low power architecture for embedded perception," in *Proc. Int. Conf. Compilers Archit. Synthesis Embedded Syst.*, 2004, pp. 46–56. [Online]. Available: http://doi.acm.org/10.1145/1023833.1023842

[42] H. Tabani, J.-M. Arnau, J. Tubella, and A. Gonzalez, "An ultra low-power hardware accelerator for acoustic scoring in speech recognition," in *Proc. 26th Int. Conf. Parallel Archit. Compilation Techn.*, 2017.

[43] G. Hinton, et al., "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Process. Mag.*, vol. 29, no. 6, pp. 82–97, Nov. 2012.

[44] P. Swietojanski, A. Ghoshal, and S. Renals, "Revisiting hybrid and GMM-HMM system combination techniques," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, 2013, pp. 6744–6748.

[45] S. P. Rath, K. M. Knill, A. Ragni, and M. J. Gales, "Combining tandem and hybrid systems for improved speech recognition and keyword spotting on low resource languages," in *Proc. Annu. Conf. Int. Speech Commun. Assoc.*, 2014, pp. 835–839.

[46] D. P. Ellis, R. Singh, and S. Sivadas, "Tandem acoustic modeling in large-vocabulary recognition," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, 2001, pp. 517–520.

[47] D. Yu and M. L. Seltzer, "Improved bottleneck features using pretrained deep neural networks," in *Proc. Annu. Conf. Int. Speech Commun. Assoc.*, 2011, Art. no. 240.

[48] D. Yu and L. Deng, *Automatic Speech Recognition: A Deep Learning Approach*. Berlin, Germany: Springer, 2014.

[49] Y. Tachioka and T. Narita, "Optimal automatic speech recognition system selection for noisy environments," in *Proc. Asia-Pacific Signal Inf. Process. Assoc. Annu. Summit Conf.*, 2016, pp. 1–8.

Hamid Tabani received the MSc degree from the University of Tehran. He joined the ARCO research group in November 2014 and started the PhD degree in the Computer Architecture Department, Universitat Politècnica de Catalunya (UPC), Barcelona, Spain. His PhD thesis focuses on low-power architectures for automatic speech recognition. His research interests include microarchitecture, cognitive computing, and new technologies.

Jose-Maria Arnau received the PhD degree on computer architecture from the Universitat Politècnica de Catalunya (UPC), in 2015. He is a member of the ARCO (ARchitecture and COmpilers) research group with UPC. His research interests include low-power architectures for cognitive computing, especially in the area of automatic speech recognition, and object recognition. He can be contacted at jarnau@ac.upc.edu.

Jordi Tubella received the degree in computer science, in 1986 and the PhD degree in computer science, in 1996, both from the Universitat Politècnica de Catalunya, Barcelona, Spain. He has been a member of the Computer Architecture Department, Universitat Politècnica de Catalunya since 1988, being an associate professor since 1998. His research interests focus on processor microarchitecture and parallel processing, with special interest on heterogeneous computing and speech recognition.

Antonio González received the PhD degree, in 1989. He is a full professor in the Computer Architecture Department, Universitat Politècnica de Catalunya, Barcelona, Spain, and was the founding director of the Intel Barcelona Research Center from 2002 to 2014. His research has focused on computer architecture, compilers and parallel processing, with a special emphasis on processor microarchitecture and code generation. He has published more than 325 papers, has given more than 100 invited talks, holds more than 40 patents, and has advised 32 PhD theses in these areas. He also has a long track record of innovations in commercial products, especially Intel products, during his stage as director of the Intel Barcelona Research Center. He has served as an associate editor of five IEEE and ACM journals, program chair for ISCA, MICRO, HPCA, ICS, and ISPASS, general chair for MICRO and HPCA, and PC member for more than 100 symposia. His awards include the award to the best student in computer engineering in Spain graduating in 1986, the 2001 Rosina Ribalta award as the advisor of the best PhD project in Information Technology and Communications, the 2008 Duran Farrell award for research in technology, the 2009 Aritmel National Award of Informatics to the Computer Engineer of the Year, the 2013 King Jaime I Award in New Technologies, and the 2014 ICREA Academia Award. He is a fellow of the IEEE.